Face Recognition Vendor Test
FRVT 2018

Performance of Automated Face Identification Algorithms

Concept, Evaluation Plan and API

VERSION 0.5

THIS DOCUMENT IS OPEN FOR COMMENTS: EMAIL TO FRVT@NIST.GOV

Patrick Grother
Mei Ngan
Kayee Hanaoka
Information Access Division
Information Technology Laboratory

October 26, 2017

National Institute of
Standards and Technology
U.S. Department of Commerce

Table of Contents

1. FRVT 2018 3
1.1. Scope 3
1.2. Audience 3
1.3. Schedule 3
1.4. Reporting 3
1.5. Version Control 3
1.6. Background 4
1.7. FRVT 2018: Changes from prior evaluations 4
1.8. Relation to the 1:1 FRVT evaluation 4
1.9. Core accuracy metrics 4
1.10. Application relevance 5
2. Rules for participation 5
2.1. Participation agreement 5
2.2. Validation 5
2.3. Hardware specification 5
2.4. Operating system, compilation, and linking environment 6
2.5. Software and documentation 6
2.6. Runtime behavior 7
2.7. Time limits 8
2.8. Template size limits 8
3. Data structures supporting the API 9
3.1. Requirement 9
3.2. File formats and data structures 9
4. API specification 13
4.1. Namespace 13
4.2. Overview 13
4.3. API 15

List of Tables

Table 1 – Schedule and allowed number of submissions 3
Table 3 – Processing time limits in seconds, per 640 x 480 color image, on a single CPU 8
Table 4 – Structure for a single image 9
Table 5 – Labels describing categories of Images 9
Table 6 – Structure for a set of images from a single person 9
Table 7 – Structure for a pair of eye coordinates 10
Table 8 – Labels describing template role 10
Table 9 – Enrollment dataset template manifest 11
Table 10 – Labels describing gallery composition 12
Table 11 – Structure for a candidate 12
Table 12 – Enumeration of return codes 12
Table 13 – ReturnStatus structure 13
Table 16 – Procedural overview of the 1:N test 13
Table 17 – Template creation initialization 15
Table 19 – Enrollment finalization 16
Table 21 – Identification search 17
Table 23 – Insertion of template into a gallery 18
Table 24 – Removal of template from a gallery 18

1. FRVT 2018

1.1. Scope

This document establishes a concept of operations and an application programming interface (API) for evaluation of one-to-many face recognition algorithms applied to faces appearing in 2D still photographs. The primary focus of the test is cooperative portrait images, e.g. mugshots[1]. The test will include also search of non-cooperative images. The API communicates the type of image to the algorithm.

1.2. Audience

Participation is open to any organization worldwide, primarily researchers and developers of FR algorithms. While NIST intends to evaluate stable technologies that could be readily made operational, the test is also open to experimental, prototype and other technologies. All algorithms **must** be submitted as implementations of the API defined in this document. There is no charge for participation.

1.3. Schedule

In consultation with US Government collaborators, NIST will execute the FRVT 2018 on the schedule given in Table 1. Note that NIST will report results publicly at the end of Phases 2 and 3. The end of Phase 2 corresponds closely to the end of the financial year and results from the FRVT are required by that date.

Developers may submit the number of algorithms identified in the rightmost column.

Table 1 – Schedule and allowed number of submissions

Phase	Date	Milestone	Maximum number of implementations
API Development	2017-10-26	Draft evaluation plan available for public comments	
	2017-11-16	Final evaluation plan published	
Phase 1	2018-01-22	Participation starts: Algorithms may be sent to NIST	
	2018-02-16	**Last day for submission of algorithms to Phase 1**	**3**
	2018-05-23	Interim results released	
Phase 2	2018-06-21	**Last day for submission of algorithms to Phase 2**	**2**
	2018-09-20	Results released	
Phase 3	2018-10-19	**Last day for submission of algorithms to Phase 3**	**2**
	2018-12-19	Release of final public report	

1.4. Reporting

At the conclusion of Phase 1, NIST will provide results for all algorithms to all developers, and to US Government partners.

At the conclusion of Phase 2, NIST will publish results for all algorithms on its website.

At the conclusion of Phase 3, NIST will publish an Interagency Report summarizing the entire FRVT 1:N.

Important: This is an open test in which NIST will identify the algorithm and the developing organization. Algorithm results will be attributed to the developer. Results will be machine generated (i.e. scripted) and will include timing, accuracy and other performance results. These will be posted alongside results from other implementations.

NIST may additionally report results in workshops, conferences, conference papers and presentations, journal articles and technical reports.

1.5. Version Control

Developers must submit a version.txt file in the doc/ folder that accompanies their algorithm – see Section 2.5.3. The string in this file should allow the developer to associate results that appear in NIST reports with the submitted algorithm.

[1] The MEDS database includes sample mugshots.

This is intended to allow end-users to obtain productized versions of the prototypes submitted to NIST. NIST will publish the contents of version.txt. NIST has previously published MD5 hashes of the core libraries for this purpose.

1.6. Background

NIST has conducted evaluations of face recognition prototypes since the first FRVT in 2000. Until 2006, those trials simulated one-to-many search accuracy using sets of one-to-one comparisons. In 2010, in support of the FBI's procurement of a face search capability, NIST reported [NISTIR 7709] accuracy and speed of end-to-end one-to-many search implementations with enrolled populations up to 1.8 million. Using updated algorithms, that test was repeated and extended in 2013 [NIST 8009]. Those tests focused on cooperative portrait images. In late 2017, NIST will publish the results from the Face Recognition Prize Challenge which assessed capability of contemporary search algorithms on less constrained images [FRPC 2017].

1.7. FRVT 2018: Changes from prior evaluations

Given massive changes in face recognition since the last one-to-many evaluation in 2013, NIST seeks to assess benefits that have accrued to the use of those technologies with cooperative images.

The following are new aspects:

- **Effect of increased population size**: N is expected to exceed 10^7 representing an order of magnitude increasing in the number of unique faces.
- **Accuracy with encounter-based galleries**: Prior NIST evaluations constructed galleries in which all known images of an enrollee were provided to the algorithm together, under a single identity. This supported algorithms which might fuse images or templates. However, this subject-based enrollment differs from that (minority of) operational deployments – encounter-based applications - in which multiple images of a person are present in a database without any specified identity or link. This test will compare accuracy for both subject-based and encounter-based galleries. Section 3.2.6 contains narrative and API support to inform the algorithm of the gallery type.
- **Expense of deletion and insertion functions**: Prior NIST evaluations constructed a gallery which was searched without any modification. This facilitated measurement of accuracy and speed but did not allow for measurement of computational expense of adding or deleting elements from the gallery. These functions may not be trivial, for example, if the underlying implementation uses fast-search data structures.

1.8. Relation to the 1:1 FRVT evaluation

Since February 2017, NIST has been running an ongoing evaluation of one-to-one face verification algorithms, FRVT 1:1. This allows any developer to submit algorithms at any time, once every three calendar months, thereby better aligning development and evaluation schedules. The FRVT 1:1 includes six different datasets, one of which is mugshots, similar to the primary set proposed for inclusion on the FRVT 1:N defined herein. It may benefit developers to submit their core 1:1 recognition algorithms to the FRVT 1:1 process by the FRVT 1:N deadline. If such cpu-based algorithms are submitted by January 22, we will publish results before the 1:N deadline.

Participation in the FRVT 1:1 is not required for participation in FRVT 1:N.

1.9. Core accuracy metrics

This test will execute open-universe searches. That is, some proportion of searches will not have an enrolled mate. From the candidate lists returned by algorithms, NIST will compute and report accuracy metrics, primarily:

- False negative identification rate (FNIR) – the proportion of mated searches which do not yield a mate within the top R ranks and at or above threshold, T.
- False positive identification rate (FPIR) – the proportion of non-mated searches returning any (1 or more) candidates at or above a threshold, T.
- Selectivity – the number of non-mated candidates returned at or above a threshold, T. This quantity has a value running from 0 to L, the number of candidates requested. It may be fractional, as it is estimated as a count divided by the number of non-mate searches.

These quantities are estimated from candidate lists produced by requesting the top L most similar candidates to the search. We do not intend to execute searches requesting only those candidates above a specified input threshold.

We will report FNIR and FPIR by sweeping the threshold over the interval [0, infinity). Error tradeoff plots (FNIR vs. FPIR, parametric on threshold) will be the primary reporting mechanism.

We will report also FNIR by sweeping a rank R over the interval [1, L] to produce (the complement of) the cumulative match characteristic (CMC).

1.10. Application relevance

NIST anticipates reporting FNIR in two FPIR regimes:

- Investigation mode: Given candidate lists and a threshold of zero, the CMC metric is relevant to investigational applications where human examiners will adjudicate candidates in decreasing order of similarity. This is common in law enforcement "lead generation".
- Identification mode: We will apply (high) thresholds to candidate lists and report FNIR values relevant to identification applications where human labor is matched to the tolerable number of false positives per unit time. This is used in duplicate detection searches for credential issuance and, more so, in surveillance applications.

Given that multiple algorithms may be submitted, developers are encouraged to submit variants tailored to minimize FNIR in the two FPIR regimes, and to explore the speed-accuracy trade space.

2. Rules for participation

2.1. Participation agreement

A participant must properly follow, complete, and submit the FRVT Participation Agreement. This must be done once, either prior or in conjunction with the very first algorithm submission. It is not necessary to do this for each submitted implementation thereafter.

NOTE Organizations that have already submitted a participation agreement for ongoing FRVT 1:1 do not need to send in a new participation agreement, as their submission of a 1:N algorithm indicates they agree to the same terms and conditions articulated in that agreement.

NOTE If an organization updates their cryptographic signing key, they must send a new completed participation agreement submission for this evaluation, with the fingerprint of their public key.

2.2. Validation

All participants must run their software through the provided FRVT 1:N validation package prior to submission. The validation package will be made available at https://github.com/usnistgov/frvt. The purpose of validation is to ensure consistent algorithm output between the participant's execution and NIST's execution.

2.3. Hardware specification

NIST intends to support high performance by specifying the runtime hardware beforehand. There are several types of computer blades that may be used in the testing. Each machine has at least 192 GB of memory. We anticipate that 16 processes can be run without time slicing, though NIST will handle all multiprocessing work via `fork()`[2]. Participant-initiated multiprocessing is not permitted.

All implementations shall use 64 bit addressing.

NIST intends to support highly optimized algorithms by specifying the runtime hardware. There are several types of computers that may be used in the testing. The following list gives some details about possible compute architectures:

[2] http://man7.org/linux/man-pages/man2/fork.2.html

- Dual Intel® Xeon® X5690 3.47 GHz CPUs (6 cores each)[3]
- Dual Intel® Xeon® CPU E5-2630 v4 @ 2.2GHz (10 cores each)[4]
- Dual Intel® Xeon® CPU E5-2680 v4 @ 2.4GHz (14 cores each)[4]

This test will not support the use of Graphics Processing Units (GPUs). The FRVT 1:1 activity, which remains open, documents relative GPU vs CPU speed.

2.4. Operating system, compilation, and linking environment

The operating system that the submitted implementations shall run on will be released as a downloadable file accessible from http://nigos.nist.gov:8080/evaluations/CentOS-7-x86_64-Everything-1511.iso, which is the 64-bit version of CentOS 7.2 running Linux kernel 3.10.0.

For this test, MacOS and Windows-compiled libraries are not permitted. All software must run under CentOS 7.2.

NIST will link the provided library file(s) to our C++ language test drivers. Participants are required to provide their library in a format that is dynamically-linkable using the C++11 compiler, g++ version 4.8.5.

A typical link line might be

```
g++ -std=c++11 -I. -Wall -m64 -o frvt1N frvt1N.cpp -L. –lfrvt1N_acme_0
```

The Standard C++ library should be used for development. The prototypes from this document will be written to a file "frvt1N.h" which will be included via #include.

The header files will be made available to implementers at https://github.com/usnistgov/frvt. All algorithm submissions will be built against the officially published header files – developers should not alter the header files when compiling and building their libraries.

All compilation and testing will be performed on x86_64 platforms. Thus, participants are strongly advised to verify library-level compatibility with g++ (on an equivalent platform) prior to submitting their software to NIST to avoid linkage problems later on (e.g. symbol name and calling convention mismatches, incorrect binary file formats, etc.).

2.5. Software and documentation

2.5.1. Library and platform requirements

Participants shall provide NIST with binary code only (i.e. no source code). The implementation should be submitted in the form of a dynamically-linked library file.

The core library shall be named according to Table 2. Additional supplemental libraries may be submitted that support this “core” library file (i.e. the “core” library file may have dependencies implemented in these other libraries). Supplemental libraries may have any name, but the “core” library must be dependent on supplemental libraries in order to be linked correctly. The **only** library that will be explicitly linked to the FRVT test driver is the “core” library.

Intel Integrated Performance Primitives (IPP) ® libraries are permitted if they are delivered as a part of the developer-supplied library package. It is the provider’s responsibility to establish proper licensing of all libraries. The use of IPP libraries shall not prevent running on CPUs that do not support IPP. Please take note that some IPP functions are multithreaded and threaded implementations are prohibited.

NIST will report the size of the supplied libraries.

[3] cat /proc/cpuinfo returns fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 cx16 xtpr pdcm pcid dca sse4_1 sse4_2 popcnt aes lahf_lm ida arat dtherm tpr_shadow vnmi flexpriority ept vpid

[4] cat /proc/cpuinfo returns fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 fma cx16 xtpr pdcm pcid dca sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch ida arat epb pln pts dtherm tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm cqm rdseed adx smap xsaveopt cqm_llc cqm_occup_llc

198 **Table 2 – Implementation library filename convention**

Form	libfrvt1N_provider_sequence.ending			
Underscore delimited parts of the filename	libfrvt1N	provider	sequence	ending
Description	First part of the name, required to be this.	Single word, non-infringing name of the main provider EXAMPLE: Acme	A one digit decimal identifier to **start at 0** and incremented by 1 for each distinct algorithm sent to NIST. Do not increment this number when submitting bug-fixed updates.	.so
Example	libfrvt1N_acme_0.so			

199 ### 2.5.2. Configuration and developer-defined data

200 The implementation under test may be supplied with configuration files and supporting data files. These might include,
201 for example, model, calibration or background feature data. NIST will report the size of the supplied configuration files.

202 ### 2.5.3. Submission folder hierarchy

203 Participant submissions shall contain the following folders at the top level

204 — lib/ - contains all participant-supplied software libraries

205 — config/ - contains all configuration and developer-defined data

206 — doc/ - contains version.txt, which documents versioning information for the submitted software and any other
207 participant-provided documentation regarding the submission

208 — validation/ - contains validation output

209 ### 2.5.4. Installation and usage

210 The implementation shall be installable using simple file copy methods. It shall not require the use of a separate
211 installation program and shall be executable on any number of machines without requiring additional machine-specific
212 license control procedures or activation. The implementation shall not use nor enforce any usage controls or limits based
213 on licenses, number of executions, presence of temporary files, etc. The implementation shall remain operable for at
214 least twelve months from the submission date.

215 ## 2.6. Runtime behavior

216 ### 2.6.1. Modes of operation

217 Implementations shall not require NIST to switch “modes” of operation or algorithm parameters. For example, the use of
218 two different feature extractors must either operate automatically or be split across two separate library submissions.

219 ### 2.6.2. Interactive behavior, stdout, logging

220 The implementation will be tested in non-interactive “batch” mode (i.e. without terminal support). Thus, the submitted
221 library shall:

222 — Not use any interactive functions such as graphical user interface (GUI) calls, or any other calls which require terminal
223 interaction e.g. reads from “standard input”.

224 — Run quietly, i.e. it should not write messages to "standard error" and shall not write to “standard output”.

225 — Only if requested by NIST for debugging, include a logging facility in which debugging messages are written to a log
226 file whose name includes the provider and library identifiers and the process PID.

2.6.3. Exception handling

The application should include error/exception handling so that in the case of a fatal error, the return code is still provided to the calling application.

2.6.4. External communication

Processes running on NIST hosts shall not side-effect the runtime environment in any manner, except for memory allocation and release. Implementations shall not write any data to external resource (e.g. server, file, connection, or other process), nor read from such, nor otherwise manipulate it. If detected, NIST will take appropriate steps, including but not limited to, cessation of evaluation of all implementations from the supplier, notification to the provider, and documentation of the activity in published reports.

2.6.5. Stateless behavior

All components in this test shall be stateless, except as noted. This applies to face detection, feature extraction and matching. Thus, all functions should give identical output, for a given input, independent of the runtime history. NIST will institute appropriate tests to detect stateful behavior. If detected, NIST will take appropriate steps, including but not limited to, cessation of evaluation of all implementations from the supplier, notification to the provider, and documentation of the activity in published reports.

2.6.6. Single-thread requirement and parallelization

Implementations must run in single-threaded mode, because NIST will parallelize the test by dividing the workload across many cores and many machines. Implementations must ensure that there are no issues with their software being parallelized via the `fork()` function.

2.7. Time limits

The elemental functions of the implementations shall execute under the time constraints of Table 3. These time limits apply to the function call invocations defined in section 4. Assuming the times are random variables, NIST cannot regulate the maximum value, so the time limits are 90-th percentiles. This means that 90% of all operations should take less than the identified duration. Timing will be estimated from at least 1000 separate invocations of each elemental function.

The time limits apply per image.

Table 3 – Processing time limits in seconds, per 640 x 480 color image, on a single CPU

Function	1:N
Template Generation	1
1:N finalization (on gallery of 1 million enrolled templates)	40000
1:N search for: – N = 1 million enrolled templates – L = 100 returned candidates	25

2.8. Template size limits

NIST anticipates evaluating performance with N well in excess of 10^7. For implementations that represent a gallery in memory with a linear data structure, the memory of our machines implies a limit on template sizes. Thus, for a template size B, the total memory requirement would be about NB. NIST anticipates running the largest N values on machines equipped with 768GB or memory. With N = 25 million, templates should not exceed 32KB.

The API, however, supports multi-stage searches and read access of the disk during the 1:N search. Disk access would likely be very slow. In all cases, algorithms shall conform to the search duration limits given in Table 3, with linear scaling.

3. Data structures supporting the API

3.1. Requirement

FRVT 1:N participants shall implement the relevant C++ prototyped interfaces of section 4. C++ was chosen in order to make use of some object-oriented features.

3.2. File formats and data structures

3.2.1. Overview

In this face recognition test, an individual is represented by $K \geq 1$ two-dimensional facial images. Most images will contain exactly one face. In a small fraction of the images, other, smaller, faces will appear in the background. Algorithms should detect one foreground face (the biggest one) in each image and produce one template.

Table 4 – Structure for a single image

C++ code fragment	Remarks
`typedef struct Image`	
`{`	
`uint16_t width;`	Number of pixels horizontally
`uint16_t height;`	Number of pixels vertically
`uint16_t depth;`	Number of bits per pixel. Legal values are 8 and 24.
`std::shared_ptr<uint8_t> data;`	Managed pointer to raster scanned data. Either RGB color or intensity. If image_depth == 24 this points to 3WH bytes RGBRGBRGB... If image_depth == 8 this points to WH bytes IIIIIII
`Label description;`	Single description of the image. The allowed values for this field are specified in the enumeration in Table 5.
`} Image;`	

An **Image** will be accompanied by one of the labels given below. Face recognition implementations should tolerate **Images** of any category.

Table 5 – Labels describing categories of Images

Label as C++ enumeration	Meaning
`enum class Label {`	
`UNKNOWN=0,`	Either the label is unknown or unassigned.
`ISO,`	Frontal, intended to be in conformity to ISO/IEC 19794-5:2005.
`MUGSHOT,`	From law enforcement booking processes. Nominally frontal.
`PHOTOJOURNALISM,`	The image might appear in a news source or magazine. The images are typically taken by professional photographer and are well exposed and focused but exhibit pose and illumination variations.
`EXPLOITATION,`	The image is taken from a child exploitation database. This imagery has highly unconstrained pose and illumination, expression and resolution.
`WILD`	Unconstrained image, taken by an amateur photographer, exhibiting wide variations in pose, illumination, and resolution.
`};`	

Table 6 – Structure for a set of images from a single person

C++ code fragment	Remarks
`using Multiface = std::vector<Image>;`	Vector of Image objects

3.2.2. Data structure for eye coordinates

Implementations shall return eye coordinates of each facial image. This function, while not necessary for a recognition test, will assist NIST in assuring the correctness of the test database. The primary mode of use will be for NIST to inspect images for which eye coordinates are not returned, or differ between implementations.

The eye coordinates shall follow the placement semantics of the ISO/IEC 19794-5:2005 standard - the geometric midpoints of the endocanthion and exocanthion (see clause 5.6.4 of the ISO standard).

Sense: The label "left" refers to subject's left eye (and similarly for the right eye), such that xright < xleft.

Table 7 – Structure for a pair of eye coordinates

C++ code fragment	Remarks
`typedef struct EyePair`	
`{`	
`bool isLeftAssigned;`	If the subject's left eye coordinates have been computed and assigned successfully, this value should be set to true, otherwise false.
`bool isRightAssigned;`	If the subject's right eye coordinates have been computed and assigned successfully, this value should be set to true, otherwise false.
`uint16_t xleft;` `uint16_t yleft;`	X and Y coordinate of the center of the subject's left eye. If the eye coordinate is out of range (e.g. x < 0 or x >= width), `isLeftAssigned` should be set to false.
`uint16_t xright;` `uint16_t yright;`	X and Y coordinate of the center of the subject's right eye. If the eye coordinate is out of range (e.g. x < 0 or x >= width), `isRightAssigned` should be set to false.
`} EyePair;`	

3.2.3. Template role

Labels describing the type/role of the template to be generated will be provided as input to template generation. This supports asymmetric algorithms where the enrollment and recognition templates may differ in content and size.

Table 8 – Labels describing template role

Label as C++ enumeration	Meaning
`enum class TemplateRole {`	
`Enrollment_1N,`	Enrollment template for 1:N identification
`Search_1N`	Search template for 1:N identification
`};`	

3.2.4. Data type for similarity scores

Identification and verification functions shall return a measure of the similarity between the face data contained in the two templates. The datatype shall be an eight-byte double precision real. The legal range is [0, DBL_MAX], where the DBL_MAX constant is larger than practically needed and defined in the <climits> include file. Larger values indicate more likelihood that the two samples are from the same person.

Providers are cautioned that algorithms that natively produce few unique values (e.g. integers on [0,127]) will be disadvantaged by the inability to set a threshold precisely, as might be required to attain a false positive identification rate of exactly 0.0001, for example.

3.2.5. File structure for enrolled template collection

To support this 1:N test, NIST will concatenate enrollment templates into a single large file, the EDB (for enrollment database). The EDB is a simple binary concatenation of proprietary templates. There is no header. There are no delimiters. The EDB may be many gigabytes in length.

This file will be accompanied by a manifest; this is an ASCII text file documenting the contents of the EDB. The manifest has the format shown as an example in Table 9. If the EDB contains N templates, the manifest will contain N lines. The fields are space (ASCII decimal 32) delimited. There are three fields. Strictly speaking, the third column is redundant.

Important: If a call to the template generation function fails, or does not return a template, NIST will include the Template ID in the manifest with size 0. Implementations must handle this appropriately.

Table 9 – Enrollment dataset template manifest

Field name	Template ID	Template Length	Position of first byte in EDB
Datatype required	std::string	uint64_t	uint64_t
Example lines of a manifest file appear to the right. Lines 1, 2, 3 and N appear.	90201744	1024	0
	person01	1536	1024
	7456433	512	2560
	...		
	subject12	1024	307200000

The EDB scheme avoids the file system overhead associated with storing millions of small individual files.

3.2.6. Gallery Type

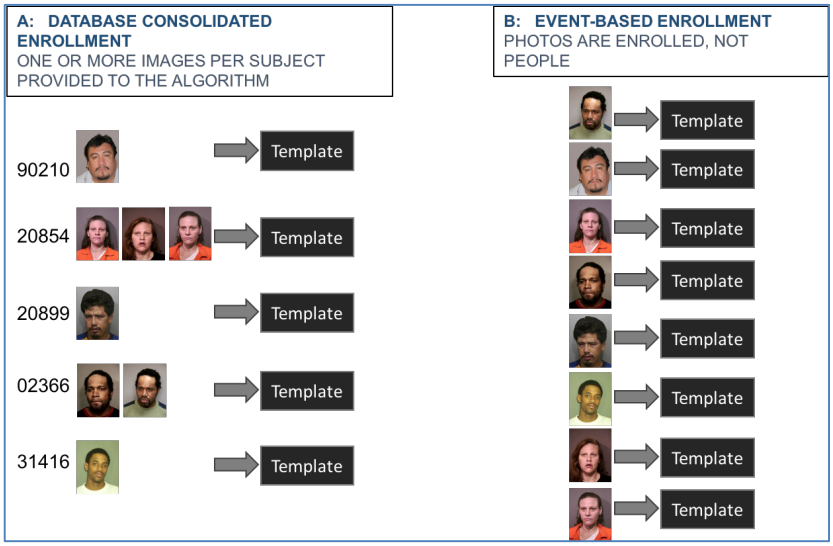

Figure 1 – Illustration of consolidated versus unconsolidated enrollment database[5]

Figure 1 illustrates two types of galleries:

- **Consolidated:** The database is formed by enrolling all images of a subject under a common identity label. The result is a gallery with N identities and N templates. This type of gallery presents us with the cleanest experimental design, "one needle in a haystack" scenario. It allows algorithms to perform image and feature level fusion. Operationally it requires high integrity biographical information to maintain.
- **Unconsolidated:** The database is formed by enrolling photographs without regard to whether the subject already has already been enrolled or not. Under this scheme, different images of the same person can exist in the gallery under different subject identifiers, that is, there are N identities, and M > N database entries.

During gallery finalization, algorithms will be provided with an enumerated label from Table 10 which specifies the type of gallery being processed.

[5] The face images contained in this figure are from the publicly available Special Database 32 - Multiple Encounter Dataset (MEDS). https://www.nist.gov/itl/iad/image-group/special-database-32-multiple-encounter-dataset-meds

319 **Table 10 – Labels describing gallery composition**

Label as C++ enumeration	Meaning
`enum class GalleryType {`	
`Consolidated,`	Consolidated, subject-based enrollment
`Unconsolidated`	Unconsolidated, event-based or photo-based enrollment
`};`	

320 ### 3.2.7. Data structure for result of an identification search

321 All identification searches shall return a candidate list of a NIST-specified length. The list shall be sorted with the most
322 similar matching entries list first with lowest rank. The data structure shall be that of Table 11.

323 **Table 11 – Structure for a candidate**

	C++ code fragment	Remarks
1.	`typedef struct Candidate`	
2.	`{`	
3.	`bool isAssigned;`	If the candidate computation succeeded, this value is set to true. False otherwise. If value is set to false, similarityScore and templateId will be ignored entirely.
4.	`std::string templateId;`	The Template ID from the enrollment database manifest defined in clause 3.2.5.
5.	`double similarityScore;`	Measure of similarity between the identification template and the enrolled candidate. Higher scores mean more likelihood that the samples are of the same person. An algorithm is free to assign any value [0, DBL_MAX] to a candidate. The distribution of values will have an impact on the false-negative and false-positive identification rates.
6.	`} Candidate;`	

324 ### 3.2.8. Data structure for return value of API function calls

325 **Table 12 – Enumeration of return codes**

Return code as C++ enumeration	Meaning
`enum class ReturnCode {`	
`Success=0,`	Success
`ConfigError,`	Error reading configuration files
`RefuseInput,`	Elective refusal to process the input, e.g. because cannot handle greyscale
`ExtractError,`	Involuntary failure to process the image, e.g. after catching exception
`ParseError,`	Cannot parse the input data
`TemplateCreationError,`	Elective refusal to produce a “non-blank” template (e.g. insufficient pixels between the eyes)
`VerifTemplateError,`	For matching, either or both of the input templates were result of failed feature extraction
`FaceDetectionError,`	Unable to detect a face in the image
`NumDataError,`	The implementation cannot support the number of images
`TemplateFormatError,`	Template file is in an incorrect format or defective
`EnrollDirError,`	An operation on the enrollment directory failed (e.g. permission, space)
`InputLocationError,`	Cannot locate the input data – the input files or names seem incorrect
`MemoryError,`	Memory allocation failed (e.g. out of memory)
`NotImplemented,`	Function is not implemented
`VendorError,`	Vendor-defined failure. Vendor errors shall return this error code and document the specific failure in the ReturnStatus.info string from Table 13.
`};`	

326
327
328

329 **Table 13 – ReturnStatus structure**

C++ code fragment	Meaning
`struct ReturnStatus {`	
`ReturnCode code;`	Return Code
`std::string info;`	Optional information string
`// constructors`	
`};`	

330

331 4. API specification

332 Please note that included with the FRVT 1:N validation package (available at https://github.com/usnistgov/frvt) is a "null"
333 implementation of this API. The null implementation has no real functionality but demonstrates mechanically how one
334 could go about implementing this API.

335 4.1. Namespace

336 All data structures and API interfaces/function calls will be declared in the `FRVT` namespace.

337 4.2. Overview

338 The 1:N identification application proceeds in three phases: enrollment, finalization and identification. The identification
339 phase includes separate probe feature extraction and search stages.

340 The design reflects the following *testing* objectives for 1:N implementations.

- support distributed enrollment on multiple machines, with multiple processes running in parallel
- allow recovery after a fatal exception, and measure the number of occurrences
- allow NIST to copy enrollment data onto many machines to support parallel testing
- respect the black-box nature of biometric templates
- extend complete freedom to the provider to use arbitrary algorithms
- support measurement of duration of core function calls
- support measurement of template size
- support measurement of template insertion and removal times into an enrollment database

341 **Table 14 – Procedural overview of the 1:N test**

Phase	#	Name	Description	Performance Metrics to be reported by NIST
Enrollment	E1	Initialization	**initializeTemplateCreation(TemplateRole=Enrollment_1N)** Give the implementation the name of a directory where any provider-supplied configuration data will have been placed by NIST. This location will otherwise be empty. The implementation is permitted **read-only** access to the configuration directory.	
	E2	Parallel Enrollment	**createTemplate(TemplateRole=Enrollment_1N)** For each of N individuals, pass K = 1 image of the individual to the implementation for conversion to a template. The implementation will return a template to the calling application. NIST's calling application will be responsible for storing all templates as binary files. These will not be available to the implementation during this enrollment phase. Multiple instances of the calling application may run simultaneously or sequentially. These may be executing on different computers.	Statistics of the times needed to enroll an individual. Statistics of the sizes of created templates. The incidence of failed template creations.
	E3	Finalization	**finalizeEnrollment()** Permanently finalize the enrollment directory. This supports, for example, adaptation	Size of the enrollment database as a function of

			of the image-processing functions, adaptation of the representation, writing of a manifest, indexing, and computation of statistical information over the enrollment dataset. The implementation is permitted **read-write-delete access** to the enrollment directory during this phase.	population size N. Duration of this operation. The time needed to execute this function shall be reported with the preceding enrollment times.
Probe Template Creation	S1	Initialization	**initializeTemplateCreation(TemplateRole=Search_1N)** Give the implementation the name of a directory where any provider-supplied configuration data will have been placed by NIST. This location will otherwise be empty. The implementation is permitted **read-only** access to the configuration directory.	Statistics of the time needed for this operation.
	S2	Template preparation	**createTemplate(TemplateRole=Search_1N)** For each probe, create a template from K >= 1 image. The result of this step is a search template. Multiple instances of the calling application may run simultaneously or sequentially. These may be executing on different computers.	Statistics of the time needed for this operation. Statistics of the size of the search template.
Search	S3	Initialization	**initializeIdentification()** Tell the implementation the location of an enrollment directory. The implementation should read all or some of the enrolled data into main memory, so that searches can commence. The implementation is permitted **read-only access** to the enrollment directory during this phase.	Statistics of the time needed for this operation.
	S4	Search	**identifyTemplate ()** A template is searched against the enrollment database. Developers shall not attempt to improve the duration of the identifyTemplate() function by offloading any of its processing into the createTemplate() function.	Statistics of the time needed for this operation. Accuracy metrics - Type I + II error rates. Failure rates.
Gallery Insert and Remove	G1	Initialization	**initializeIdentification ()** Tell the implementation the location of an enrollment directory. The implementation should read all or some of the enrolled data into main memory, so that searches can commence. The implementation is permitted **read-only access** to the enrollment directory during this phase.	Statistics of the duration of the operation.
	G2	Template insertion/removal into/from gallery	**galleryInsertID () / galleryDeleteID()** galleryInsertID() is executed one or more times to insert a template created with createTemplate(TemplateRole=Enrollment_1N) into the gallery. galleryDeleteID() is executed one or more times to remove a template from the gallery.	Statistics of the duration of the operation.
	G3	Search	**identifyTemplate()** A template is searched against the enrollment database.	Statistics of the duration of the operation. Accuracy metrics - Type I + II error rates.

342

343

4.3. API

4.3.1. Interface

The software under test must implement the interface `IdentInterface` by subclassing this class and implementing each method specified therein.

	C++ code fragment	Remarks
1.	`Class IdentInterface`	
2.	`{` `public:`	
3.	`static std::shared_ptr<IdentInterface> getImplementation();`	Factory method to return a managed pointer to the `IdentInterface` object. This function is implemented by the submitted library and must return a managed pointer to the `IdentInterface` object.
4.	`// Other functions to implement`	
5.	`};`	

There is one class (static) method declared in `IdentInterface.getImplementation()` which must also be implemented. This method returns a shared pointer to the object of the interface type, an instantiation of the implementation class. A typical implementation of this method is also shown below as an example.

	C++ code fragment	Remarks

	`#include "frvt1N.h"`
	
	`using namespace FRVT;`
	
	`NullImpl:: NullImpl () { }`
	
	`NullImpl::~ NullImpl () { }`
	
	`std::shared_ptr<IdentInterface>`
	`IdentInterface::getImplementation()`
	`{`
	`  return std::make_shared<NullImpl>();`
	`}`
	`// Other implemented functions`	

4.3.2. Initialization of template creation

Before any feature extraction/template creation calls are made, the NIST test harness will call the initialization function of Table 15. This function will be called BEFORE any calls to fork() are made.

Table 15 – Template creation initialization

Prototype	ReturnStatus initializeTemplateCreation(		
	const std::string &configDir,		Input
	TemplateRole role);		Input
Description	This function initializes the implementation under test and sets all needed parameters in preparation for template creation. This function will be called N=1 times by the NIST application, prior to parallelizing M >= 1 calls to `createTemplate()` via `fork()`. This function will be called from a single process/thread.		
Input Parameters	configDir	A read-only directory containing any developer-supplied configuration parameters or run-time data files.	
	role	A value from the TemplateRole enumeration that indicates the intended usage of the template to be generated. In this case, either Enrollment_1N or Search_1N.	
Output Parameters	None		
Return Value	See Table 12 for all valid return code values.		

4.3.3. Template Creation

A Multiface is converted to a single template using the function of Table 16.

Table 16 – Template Creation/Feature Extraction

Prototypes	ReturnStatus createTemplate(		
	const Multiface &faces,		Input
	TemplateRole role,		Input
	std::vector<uint8_t> &templ,		Output
	std::vector<EyePair> &eyeCoordinates);		Output
Description	Takes a Multiface and outputs a proprietary template and associated eye coordinates. The vector to store the template will be initially empty, and it is up to the implementation to populate it with the appropriate data. Note: In the event that more than one face is detected in an image, features should be extracted from the foreground face, that is, the largest face in the image. *For enrollment templates (TemplateRole=Enrollment_1N)*: If the function executes correctly (i.e. returns a successful return code), the template will be enrolled into a gallery. The NIST calling application may store the resulting template, concatenate many templates, and pass the result to the enrollment finalization function (see section 4.3.4). The resulting template may also be inserted immediately into previously finalized gallery. When the implementation fails to produce a template (i.e. returns a non-successful return code), it shall still return a blank template (which can be zero bytes in length). The template will be included in the enrollment database/manifest like all other enrollment templates, but is not expected to contain any feature information. IMPORTANT: NIST's application writes the template to disk. Any data needed during subsequent searches should be included in the template, or created from the templates during the enrollment finalization function of section 4.3.4. *For identification/probe templates (TemplateRole=Search_1N)*: The NIST calling application may commit the template to permanent storage, or may keep it only in memory (the developer implementation does not need to know). If the function returns a non-successful return status, the output template will not be used in subsequent search operations.		
Input Parameters	face	Input Multiface	
	role	Label describing the type/role of the template to be generated. In this case, it will either be Enrollment_1N or Search_1N.	
Output Parameters	templ	The output template. The format is entirely unregulated. This will be an empty vector when passed into the function, and the implementation can resize and populate it with the appropriate data.	
	eyeCoordinates	The function shall return the estimated eye centers for the input face image.	
Return Value	See Table 12 for all valid return code values.		

4.3.4. Finalization

After all templates have been created, the function of Table 17 will be called. This freezes the enrollment data. After this call the enrollment dataset will be forever read-only.

The function allows the implementation to conduct, for example, statistical processing of the feature data, indexing and data re-organization. The function may alter the file structure. It may increase or decrease the size of the stored data. No output is expected from this function, except a return code.

Implementations shall not move the input data. Implementations shall not point to the input data. Implementations should not assume the input data will be readable after the call. Implementations must, at a minimum, copy the input data or otherwise extract what is needed for search.

Table 17 – Enrollment finalization

Prototypes	ReturnStatus finalizeEnrollment(	
	const std::string &enrollmentDir,	Input
	const std::string &edbName,	Input
	const std::string &edbManifestName,	Input
	GalleryType galleryType);	Input

Description	This function takes the name of the top-level directory where the enrollment database (EDB) and its manifest have been stored. These are described in section 3.2.5. The enrollment directory permissions will be read + write. The function supports post-enrollment, developer-optional, book-keeping operations, statistical processing and data re-ordering for fast in-memory searching. The function will generally be called in a separate process after all the enrollment processes are complete. This function should be tolerant of being called two or more times. Second and third invocations should probably do nothing. This function will be called from a single process/thread.	
Input Parameters	enrollmentDir	The top-level directory in which enrollment data was placed. This variable allows an implementation to locate any private initialization data it elected to place in the directory.
	edbName	The name of a single file containing concatenated templates, i.e. the EDB of section 3.2.5. While the file will have read-write-delete permission, the implementation should only alter the file if it preserves the necessary content, in other files for example. The file may be opened directly. It is not necessary to prepend a directory name. This is a NIST-provided input – implementers shall not internally hard-code or assume any values.
	edbManifestName	The name of a single file containing the EDB manifest of section 3.2.5. The file may be opened directly. It is not necessary to prepend a directory name. This is a NIST-provided input – implementers shall not internally hard-code or assume any values.
	galleryType	A label from Table 10 specifying the composition of the gallery.
Output Parameters	None	
Return Value	See Table 12 for all valid return code values.	

4.3.5. Search Initialization

The function of Table 18 will be called once prior to one or more calls of the searching function of Table 19. The function might set static internal variables so that the enrollment database is available to the subsequent identification searches. This function will be called BEFORE any calls to fork() are made.

Table 18 – Identification initialization

Prototype	ReturnStatus initializeIdentification(	
	const string &configDir,	Input
	const string &enrollmentDir);	Input
Description	This function reads whatever content is present in the enrollmentDir, for example a manifest placed there by the finalizeEnrollment() function. This function will be called from a single process/thread.	
Input Parameters	configDir	A read-only directory containing any developer-supplied configuration parameters or run-time data files.
	enrollmentDir	The read-only top-level directory in which enrollment data was placed.
Return Value	See Table 12 for all valid return code values.	

4.3.6. Search

The function of Table 19 compares a proprietary identification template against the enrollment data and returns a candidate list.

Table 19 – Identification search

Prototype	ReturnStatus identifyTemplate (	
	const std::vector<uint8_t> &idTemplate,	Input
	const uint32_t candidateListLength,	Input
	std::vector<Candidate> &candidateList,	Output
	bool &decision);	Output

Description	This function searches a template against the enrollment set, and outputs a list of candidates. The candidateList vector will initially be empty, and the implementation shall populate the vector with candidateListLength entries.	
Input Parameters	idTemplate	A template from createTemplate(TemplateRole=Search_1N) - If the value returned by that function was non-zero the contents of idTemplate will not be used and this function (i.e. identifyTemplate) will not be called.
	candidateListLength	The number of candidates the search should return
Output Parameters	candidateList	A vector containing "candidateListLength " objects of candidates. The datatype is defined in section 3.2.6. Each candidate shall be populated by the implementation. The candidates shall appear in descending order of similarity score - i.e. most similar entries appear first.
	decision	A best guess at whether there is a mate within the enrollment database. If there was a mate found, this value should be set to true, Otherwise, false. Many such decisions allow a single point to be plotted alongside a DET.
Return Value	See Table 12 for all valid return code values.	

NOTE: Ordinarily the calling application will set the input candidate list length to operationally typical values, say $0 \leq L \leq 200$, and $L << N$. We will measure the dependence of search duration on L.

4.3.7. Gallery insertion and removal of templates

The functions of this section insert a new template into, and removes an existing template from, a finalized gallery, respectively.

Table 20 – Insertion of template into a gallery

Prototype	ReturnStatus galleryInsertID(	
	const std::vector<uint8_t> &templ,	Input
	const std::string &id);	Input
Description	This function inserts a template with an associated id into an existing finalized gallery. Invocation of this function will always be preceded by a call to initializeIdentification(), which will provide the location of the finalized gallery to be loaded into memory. One or more calls to identifyTemplate() may be made after calling this function. The template ID will not exist in the database already, so a 1:N duplicate search is not necessary. This function will be called from a single process/thread.	
Input Parameters	templ	A template created via createTemplate(TemplateRole=Enrollment_1N)
	id	An identifier associated with the enrollment template
Return Value	See Table 12 for all valid return code values.	

Table 21 – Removal of template from a gallery

Prototype	ReturnStatus galleryDeleteID(	
	const std::string &id);	Input
Description	This function deletes an existing template from a finalized gallery. Invocation of this function will always be preceded by a call to initializeIdentification(), which will provide the location of the finalized gallery. One or more calls to identifyTemplate may be made before and after calling this function. The template ID will exist in the database. This function will be called from a single process/thread.	
Input Parameters	id	An identifier associated with the enrollment template
Return Value	See Table 12 for all valid return code values.	